

AD-A081 452

UNCLASSIFIED

CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/6 9/2
ON OPTIMISTIC METHODS FOR CONCURRENCY CONTROL.(U)
OCT 79 H T KUNG, J T ROBINSON
CMU-CS-79-149

N00014-76-C-0370
NL

[0]

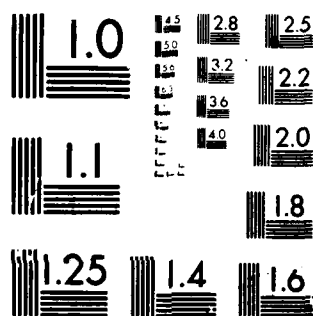
ALL INFORMATION CONTAINED HEREIN IS UNCLASSIFIED

END

DATE FILMED

4-80

DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

ADA081452

**On Optimal Methods for
Concurrency Control**

H. T. King and John T. Robinson

May 1979

(Last revised October 1979)

See 1473 in Index

**DEPARTMENT
of
COMPUTER SCIENCE**

**DTIC
ELECTE
S D**

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

Carnegie-Mellon University

80 3 5 011

DOC FILE COPY

On Optimistic Methods for Concurrency Control

H. T. Kung and John T. Robinson

May 1979

(Last revised October 1979)

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

ABSTRACT

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this paper, two families of non-locking concurrency controls are presented. The methods used are "optimistic" in the sense that they rely mainly on transaction backup as a control mechanism, "hoping" that conflicts between transactions will not occur. Applications where these methods should be more efficient than locking are discussed.

Copyright -C- 1979 by H. T. Kung and John T. Robinson

This research is supported in part by the National Science Foundation under Grant MCS 78-236-76 and the Office of Naval Research under Contract N00014-76-C-0370.

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

DTIC
ELECTE
MAR 6 1980
S B D

1. Introduction

Consider the problem of providing shared access to a data structure organized as a directed graph, i.e., a collection of nodes where each node consists of some values local to that node and some pointers to other nodes. Certain distinguished nodes, called the roots, are always present, and access to any node other than a root is gained only by first accessing a root and then following pointers to that node. Any sequence of accesses to the data structure that preserves the integrity constraints of the data is called a *transaction* (see, e.g. [5]).

If our goal is to maximize the throughput of accesses to the data structure, then there are at least two cases where highly concurrent access is desirable:

- The amount of data is sufficiently great that at any given time only a fraction of the data structure can be present in primary memory, so that it is necessary to swap parts of the data structure from secondary memory as needed.
- Even if the entire data structure can be present in primary memory, there may be multiple processors.

In both cases the hardware will be under-utilized if the degree of concurrency is too low.

However, as is well-known, unrestricted concurrent access to a shared data structure will in general cause the integrity of the data structure to be lost. Most current approaches to this problem involve some type of locking. That is, a mechanism is provided whereby one process can deny certain other processes access to some portion of the data structure. In particular, a lock may be associated with each node of the directed graph, and any given process is required to follow some locking protocol, so as to guarantee that no other process can ever discover any lack of integrity in the data structure temporarily caused by the given process.

The locking approach has the following inherent disadvantages:

1. Lock maintenance represents an overhead that is not present in the sequential case. Even read-only transactions (queries), which cannot possibly affect the integrity of the data, must in general use locking in order to guarantee that the data being read are not modified by other transactions at the same time. Also, if the locking protocol is not deadlock free, deadlock detection must be considered to be part of lock maintenance overhead. In the case of System R [1], it has been noted that lock maintenance represents 10% of total execution time [6].
2. There are no general purpose deadlock-free locking protocols for directed graph access algorithms that always provide high concurrency. Because of this, some research has been directed at developing special purpose locking protocols for various special cases of the general directed graph structure, access algorithms,

Write Section	<input checked="" type="checkbox"/>
Diff Section	<input type="checkbox"/>
	<input type="checkbox"/>

BY		
DISTRIBUTION/AVAILABILITY CODES		
Dist.	AVAIL.	and/or SPECIAL
A		

and integrity criteria. In the case of B-trees [2], at least nine locking protocols have been proposed [3, 4, 10, 11, 14].

3. In the case that large parts of the data structure are on secondary memory, concurrency is significantly lowered whenever it is necessary to leave some congested node locked (a congested node is one that is often accessed, e.g. a root) while waiting for a secondary memory access.
4. To allow a transaction to abort itself when mistakes occur, locks cannot be released until the end of the transaction. This may again significantly lower concurrency.
5. Most important for the purposes of this paper, *locking may be necessary at all only in the worst case*. Consider the following simple example: the directed graph consists solely of roots, and each transaction involves one root only, any root equally likely. Then if there are n roots and two processes executing transactions at the same rate, locking is *really* needed (if at all) every n transactions, on the average.

In general, one may expect the argument of 5) to hold whenever a) the number of nodes in the graph is very large compared to the total number of nodes involved in all the running transactions at a given time, and b) the probability of modifying a congested node is small. In many applications, a) and b) are designed to hold (see Section 6 for the B-tree application).

Research directed at finding deadlock-free locking protocols may be seen as an attempt to lower the expense of concurrency control by eliminating transaction backup as a control mechanism. In this paper we consider the converse problem, that of eliminating locking. We propose two families of concurrency controls that do not use locking. These methods are "optimistic" in the sense that they rely for efficiency on the hope that conflicts between transactions will not occur. If 5) does hold, such conflict will be rare. This approach also has the advantage that it is completely general, applying equally well to any shared directed graph structure and associated access algorithms. Since locks are not used, it is deadlock-free (however, starvation is a possible problem, a solution for which we discuss). It is also possible using this approach to avoid problem 3) and 4) above. Finally, if the transaction pattern becomes query dominant (i.e., most transactions are read-only), then the concurrency control overhead becomes almost totally negligible (a partial solution to problem 1)).

The idea behind this optimistic approach is quite simple, and may be summarized as follows:

- Since reading a value or a pointer from a node can never cause a loss of integrity, reads are completely unrestricted (however, returning a result from a query is considered to be equivalent to a write, and so is subject to validation as discussed below).

- Writes are severely restricted. It is required that any transaction consist of two or three phases: a *read phase*, a *validation phase*, and a possible *write phase*. During the read phase, all writes take place on local copies of the nodes to be modified. Then, if it can be established during the validation phase that the changes the transaction made will not cause a loss of integrity, the local copies are made global in the write phase. In the case of a query, it must be determined that the result the query would return is actually correct. The step in which it is determined that the transaction will not cause a loss of integrity (or that it will return the correct result) is called *validation*.

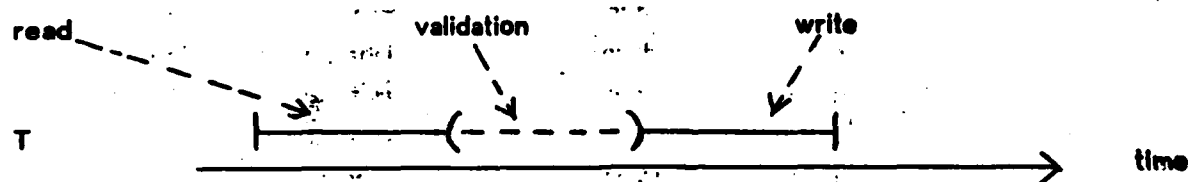


Figure 1. The three phases of a transaction T

If, in a locking approach, locking is only necessary in the worst case, then in an optimistic approach validation will fail also only in the worst case. If validation does fail, the transaction will be backed up and start over again as a new transaction. Thus, a transaction will have a write phase only if the preceding validation succeeds.

In Section 2 we discuss in more detail the read and write phases of transactions. In Section 3 a particularly strong form of validation is presented. The correctness criteria used for validation are based on the notion of serial equivalence [5, 13, 15]. In the next two sections concurrency controls are presented that rely on the serial equivalence criteria developed in Section 3 for validation. The family of concurrency controls in Section 4 have serial final validation steps, while the concurrency controls of Section 5 have completely parallel validation, at however higher total cost. In Section 6 we analyze the application of optimistic methods to controlling concurrent insertions in B-trees. Section 7 contains a summary and a discussion of future research.

2. The Read and Write Phases

In this section we briefly discuss how the concurrency control can support the read and write phases of user programmed transactions (in a manner invisible to the user), and how this can be implemented efficiently. The validation phase will be treated in the following three sections.

We assume that an underlying system provides for the manipulation of objects of various types, and that each node of the directed graph structure is an object. For simplicity, assume all nodes are objects of the same type. Objects are manipulated by the following calls of the concurrency control, where n is the name of an object, i is a parameter to the type manager, and v is a value of arbitrary type (v could be a pointer, i.e. an object name, or data):

create create a new object and return its name.
delete(n) delete object n .
read(n, i) read item i of object n and return its value.
write(n, i, v) write v as item i of object n .

In order to support the read and write phases of transactions we will also use the following calls:

copy(n) create a new object that is a copy of object n and return its name.
exchange($n1, n2$) exchange the names of objects $n1$ and $n2$.

The concurrency control is invisible to the user; transactions are written as if the above calls were used directly. However, transactions are required to use the syntactically identical calls *tcreate*, *tdelete*, *tread*, and *twrite* to the concurrency control. For each transaction, the concurrency control maintains sets of object names accessed by the transaction. These sets are initialized to be empty by a *tbegin* call. The body of the user written transaction is in fact the read phase mentioned in the introduction; the subsequent validation phase does not begin until after a *tend* call. The semantics of the calls to the concurrency control are as follows:

tcreate =
 ($n := \text{create};$
 $\text{create set} := \text{create set } U \{n\};$
 RETURN n)

twrite(n, i, v) =
 (**IF** $n \in \text{create set}$
 THEN $\text{write}(n, i, v)$
 ELSE IF $n \in \text{write set}$
 THEN $\text{write}(\text{copies}[n], i, v)$
 ELSE ($m := \text{copy}(n);$
 $\text{copies}[n] := m;$
 $\text{write set} := \text{write set } U \{n\};$
 $\text{write}(\text{copies}[n], i, v)$))


```

tread(n,i) =
  (read set := read set U {n};
   IF n ∈ write set
   THEN RETURN read(copies[n],i)
   ELSE RETURN read(n,i) )

```

```

tdelete(n) =
  (delete set := delete set U {n})

```

Above, *copies* is an associative vector of object names, indexed by object name. We see that in the read phase, no global writes take place. Instead, whenever the first write to a given object is requested, a copy is made, and all subsequent writes are directed to the copy. This copy is potentially global, but is inaccessible to other transactions during the read phase by our convention that all nodes are accessed only by following pointers from a root node. If the node is a root node, the copy is inaccessible since it has the wrong name (all transactions "know" the global names of root nodes). It is assumed that no root node is created or deleted, that no dangling pointers are left to deleted nodes, and that created nodes become accessible by writing new pointers (these conditions are part of the integrity criteria for the data structure that each transaction is required to individually preserve).

When the transaction completes, it will request its validation and write phases via a *tend* call. If validation succeeds, then the transaction enters the write phase, which is simply:

```

FOR n ∈ write set DO exchange(n, copies[n]) .

```

After the write phase all written values become "global", all created nodes become accessible, and all deleted nodes become inaccessible. Of course some cleanup is necessary, which we do not consider to be part of the write phase since it does not interact with other transactions:

```

(FOR n ∈ delete set DO delete(n);
 FOR n ∈ write set DO delete(copies[n]) ) .

```

Similar types of cleanup may be necessary for transaction backup, which we do not consider in detail here.

Note that since objects are virtual (objects are referred to by name, not by physical

address) the *exchange* operation, and hence the write phase, can be made quite fast: essentially, all that is necessary is to exchange the physical address parts of the two object descriptors.

Finally, we note that the concept of two-phase transactions appears to be quite valuable for recovery purposes, since at the end of the read phase, all changes that the transaction intends to make to the data structure are known.

3. The Validation Phase

A widely used criterion for verifying the correctness of concurrent execution of transactions has been variously called serial equivalence [5], serial reproducibility [12], and linearizability [15]. This criterion may be defined as follows:

Let transactions T_1, T_2, \dots, T_n be executed concurrently. Denote an instance of the shared data structure by d , and let D be the set of all possible d , so that each T_i may be considered as a function:

$$T_i: D \rightarrow D.$$

If the initial data structure is d_i and the final data structure is d_f , the concurrent execution of transactions is correct if some permutation π of $\{1, 2, \dots, n\}$ exists such that

$$d_f = T_{\pi(n)} \circ T_{\pi(n-1)} \circ \dots \circ T_{\pi(2)} \circ T_{\pi(1)}(d_i), \quad (1)$$

where "o" is the usual notation for functional composition.

The idea behind this correctness criterion is that, first, each transaction is assumed to have been written so as to individually preserve the integrity of the shared data structure. That is, if d satisfies all integrity criteria, then for each T_i , $T_i(d)$ satisfies all integrity criteria. Now, if d_i satisfies all integrity criteria and the concurrent execution of T_1, T_2, \dots, T_n is serially equivalent, then from (1), by repeated application of the integrity preserving property of each transaction, d_f satisfies all integrity criteria. Serial equivalence is useful as a correctness criterion since it is in general much easier to verify that a) each transaction preserves integrity and b) every concurrent execution of transactions is serially equivalent, than it is to verify directly that every concurrent execution of transactions preserves integrity. In fact, it has been shown in [8] that serialization is the weakest criterion for preserving consistency of a concurrent transaction system, even if complete syntactic information of the system is available to the concurrency control. However, if semantic information is available, then other approaches may be more attractive (see, e.g., [7, 9]).

3.1. Validation of Serial Equivalence

The use of validation of serial equivalence as a concurrency control is a direct application of condition (1) above. However, in order to verify (1), a permutation π must be found. This is handled by *explicitly* assigning each transaction T_i a unique integer *transaction number* $t(i)$ during the course of its execution. The meaning of transaction numbers in validation is the following: there must exist a serially equivalent schedule in which transaction T_j comes before transaction T_i whenever $t(j) < t(i)$. This can be guaranteed by the following validation condition: For each transaction T_i with transaction number $t(i)$, and for all T_j with $t(j) < t(i)$, one of the following three conditions must hold (see Figure 2):

1. T_j completes its write phase before T_i starts its read phase.
2. The write set of T_j does not intersect the read set of T_i , and T_j completes its write phase before T_i starts its write phase.
3. The write set of T_j does not intersect the read set or the write set of T_i , and T_j completes its read phase before T_i completes its read phase.

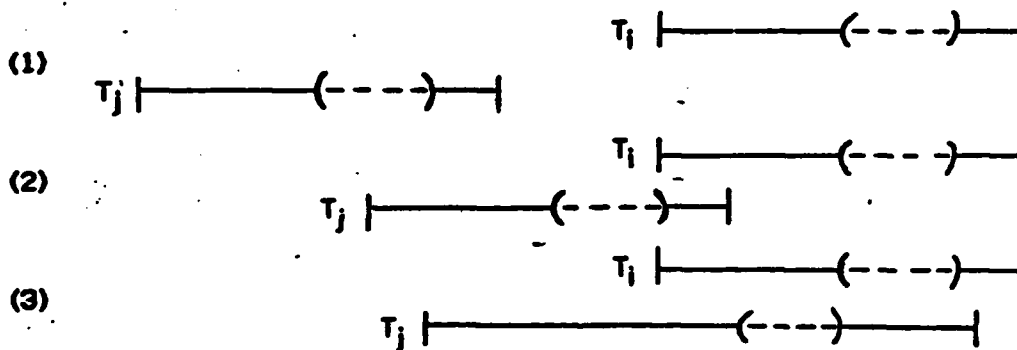


Figure 2. Possible interleaving of T_i and T_j

Condition 1) states that T_j actually completes before T_i starts. Condition 2) states that the writes of T_j do not affect the read phase of T_i , and that T_j finishes writing before T_i starts writing, hence does not overwrite T_i (also, note that T_i cannot affect the read phase of T_j). Finally, condition 3) is similar to condition 2), but does not require that T_j finish writing before T_i starts writing; it simply requires that T_j not affect the read phase or the write phase of T_i (again, note that T_i cannot affect the read phase of T_j , by the last part of the condition). See [13] for a set of similar conditions for serialization.

3.2. Assigning Transaction Numbers

The first consideration that arises in the design of concurrency controls that explicitly assign transaction numbers is, how should transaction numbers be assigned? Clearly they should somehow be assigned in order, since if T_j completes before T_i starts, we must have $t(j) < t(i)$. Here we use the simple solution of maintaining a global integer counter TNC (transaction number counter); when a transaction number is needed, the counter is incremented, and the resulting value returned. Also, transaction numbers must be assigned somewhere before validation, since the validation conditions above require knowledge of the transaction number of the transaction being validated. On first thought, we might assign transaction numbers at the beginning of the read phase; however, this is not optimistic (hence contrary to the philosophy of this paper) for the following reason. Consider the case of two transactions, T_1 and T_2 , starting at roughly the same time, assigned transaction number n and $n+1$, respectively. Even if T_2 completes its read phase much earlier than T_1 , before being validated T_2 must wait for the completion of the read phase of T_1 , since the validation of T_2 in this case relies on knowledge of the write-set of T_1 . See Figure 3. In an optimistic approach, we would like for transactions to be validated immediately if at all possible (in order to improve response time). For these and similar considerations we assign transaction numbers at the end of the read phase. Note that by assigning transaction numbers in this fashion the last part of condition 3), that T_j complete its read phase before T_i completes its read phase if $t(j) < t(i)$, is automatically satisfied.

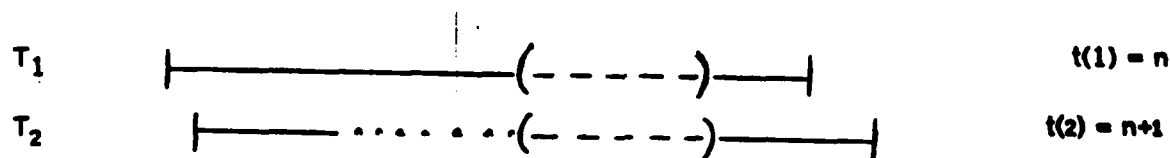


Figure 3. T_2 waits for T_1 in

3.3. Some Practical Considerations

Given this method for assigning transaction numbers, consider the case of a transaction T that has an arbitrarily long read phase. When this transaction is validated, the write sets of all transactions that completed their read phase before T but had not yet completed their write phase at the start of T must be examined. Since the concurrency control can only maintain finitely many write sets, we have a difficulty (this difficulty does not arise if transaction numbers are assigned at the beginning of the read phase). Clearly, if such

transactions are common, the assignment of transaction numbers described above is unsuitable. Of course, we take the optimistic approach and assume such transactions are very rare; still, a solution is needed. We solve this problem by only requiring the concurrency control to maintain some finite number of the most recent write sets, where the number is large enough to validate almost all transactions (we say write set a is more recent than write set b if the transaction number associated with a is greater than that associated with b). In the case of transactions like T , if old write sets are unavailable, validation fails, and the transaction is backed up (probably to the beginning). For simplicity, we present the concurrency controls of the next two sections as if potentially infinite vectors of write sets were maintained; the above convention is to be understood to apply.

One last consideration must be mentioned at this point, namely, what should be done when validation fails? It will be determined during validation exactly which objects were "dirtied", i.e., modified by transactions with transaction numbers less than the transaction number of the transaction being validated after the start of the read phase. The transaction will then be backed up to the earliest such point and continued, receiving a new transaction number at the completion of the read phase. Now a new difficulty arises: what should be done in the case that validation repeatedly fails? Under our optimistic assumptions, this should happen rarely, but we still need some method for dealing with this problem when it does occur. A simple solution is the following: we associate a lock with the transaction number counter, and this lock can be either read-locked or write-locked (read-locks lock out only write-locks; write-locks lock out all other locks). Normally, the concurrency control brackets access to the transaction number counter with a read lock-unlock pair. However, if the concurrency control detects a "starving" transaction (this could be detected by keeping track of the number of times validation for a given transaction fails), the transaction is restarted, this time using a write lock, which is not unlocked until the transaction is validated. When the write-lock is finally granted (standard techniques can be used to ensure that a write-lock does not cause starvation), all subsequent transactions will be locked out, until the "starving" transaction can run to completion.

4. Serial Validation

In this section we present a family of concurrency controls that are an implementation of validation conditions 1) and 2) of the previous section. Since we are not using condition 3), the last part of condition 2) implies that write phases must be serial. The simplest way to implement this is to place the assignment of a transaction number, validation, and the subsequent write phase all in a critical section. In the following, we bracket the critical section by "<" and ">". The concurrency control is as follows:

```

tbegin =
  (create set := empty;
   read set := empty;
   write set := empty;
   delete set := empty;
   start tn := TNC)

tend =
  ( <finish tn := TNC;
    valid := TRUE;
    FOR t FROM start tn+1 TO finish tn DO
      IF (write set of transaction with trans. no. t
          intersects read set)
        THEN valid := FALSE;
    IF valid
      THEN ((write phase); TNC:=TNC+1; tn:=TNC) >;
    IF valid
      THEN (cleanup)
      ELSE (backup) )

```

In the above, the transaction is assigned a transaction number via the sequence $TNC := TNC+1; tn := TNC$. An optimization has been made in that transaction numbers are assigned only if validation is successful. We may imagine that the transaction is "tentatively" assigned a transaction number of $TNC+1$ with the statement $finish\ tn := TNC$, but that if validation fails, this transaction number is freed for use by another transaction. By condition 1) of Section 3, we need not consider transactions that have completed their write phase before the start of the read phase of the current transaction. This is implemented by reading TNC in $tbegin$; since a "real" assignment of a transaction number takes place only after the write phase, it is guaranteed at this point that all transactions with transaction numbers less than or equal to $start\ tn$ have completed their write phase.

The above is perfectly suitable in the case that there is one CPU and that the write phase can usually take place in primary memory. If the write phase often cannot take place in primary memory, we probably want to have concurrent write phases, unless the write phase is still extremely short compared to the read phase (which may be the case). The concurrency controls of the next section are appropriate for this. If there are multiple CPU's, we may wish to introduce more potential parallelism in the validation step (this is only necessary for efficiency if the processors cannot be kept busy with read phases, i.e. if validation is not extremely short as compared to the read phase). This can be done by using the solution of the next section, or by the following method. At the end of the read phase, we immediately read TNC before entering the critical section, and assign this value to $mid\ tn$.

It is then known that at this point the write sets of transactions *start tn+1*, *start tn+2*, ..., *mid tn* must certainly be examined in the validation step, and this can be done outside the critical section. The concurrency control is thus:

```
tend =
  (mid tn := TNC;
   valid := TRUE;
   FOR t FROM start tn+1 TO mid tn DO
     IF (write set of transaction with trans. no. t
         intersects read set)
       THEN valid := FALSE;
   <finish tn := TNC;
   FOR t FROM mid tn+1 TO finish tn DO
     IF (write set of transaction with trans. no. t
         intersects read set)
       THEN valid := FALSE;
   IF valid
     THEN ((write phase); TNC:=TNC+1; tn:=TNC) >;
   IF valid
     THEN (cleanup)
     ELSE (backup) )
```

The above optimization can be carried out a second time: at the end of the preliminary validation step we read *TNC* a third time, and then, still outside the critical section, check the write sets of those transactions with transaction numbers from *mid tn+1* to this most recent value of *TNC*. Repeating this process, we derive a family of concurrency controls with varying numbers of stages of validation and degrees of parallelism, all of which however have a final indivisible validation step and write phase. The idea is to remove varying parts of the work done in the critical section outside the critical section, allowing greater parallelism.

Until now we have not considered the question of read-only transactions, or queries. Since queries do not have a write phase, it is unnecessary to assign them transaction numbers. It is only necessary to read *TNC* at the end of the read phase and assign its value to *finish tn*; validation for the query then consists of examining the write sets of the transactions with transaction numbers *start tn+1*, *start tn+2*, ..., *finish tn*. This need not occur in a critical section, so the above discussion on multiple validation stages does not apply to queries. This method for handling queries also applies to the concurrency controls of the next section. Note that for query dominant systems, validation will often be trivial: it may be determined that *start tn* = *finish tn*, and validation is complete. For this type of system an optimistic approach appears ideal.

5. Parallel Validation

In this section we present a concurrency control that uses all three of the validation conditions of Section 3, thus allowing greater concurrency. We retain the optimization of the previous section, only assigning transaction numbers after the write phase if validation succeeds. As in the previous solutions, *TNC* is read at the beginning and the end of the read phase; transactions with transactions numbers *start tn+1*, *start tn+2*, ..., *finish tn* all may be checked under condition 2) of Section 3. For condition 3), we maintain a set of transaction ids *active* for transactions that have completed their read phase but have not yet completed their write phase. The concurrency control is as follows (*tbegin* is as in the previous section):

```
tend =
( <finish tn := TNC;
  finish active := (make a copy of active);
  active := active U {id of this transaction} >;
  valid := TRUE;
  FOR t FROM start tn+1 TO finish tn DO
    IF (write set of transaction with trans. no. t
        intersects read set)
      THEN valid := FALSE;
  FOR i ∈ finish active DO
    IF (write set of transaction Ti intersects
        read set or write set)
      THEN valid := FALSE;
  IF valid
    THEN ( (write phase);
          <TNC := TNC+1;
          tn := TNC;
          active := active - {id of this transaction} >;
          (cleanup) )
    ELSE (<active:=active-{id of transaction}>;
          (backup) ) )
```

In the above, at the end of the read phase *active* is the set of transactions that have been assigned "tentative" transaction numbers less than that of the transaction being validated. Note that modifications to *active* and *TNC* are placed together in critical sections so as to maintain the invariant properties of *active* and *TNC* mentioned above. Entry to the first critical section is equivalent to being assigned a "tentative" transaction number.

One problem with the above is that a transaction in the set *finish active* may invalidate the given transaction, even though the former transaction is itself invalidated. A partial solution to this is to use several stages of preliminary validation, in a way completely analogous to the

multistage validation described in the previous section. At each stage, a new value of *TMC* is read, and transactions with transaction numbers up to this value are checked. The final stage then involves accessing *active* as above. The idea is to reduce the size of *active* by performing more of the validation before adding a new transaction id to *active*.

Finally, a solution is possible where transactions that have been invalidated by a transaction in *finish active* wait for that transaction to either be invalidated, and hence ignored, or validated, causing backup.¹ However, this solution involves a much more sophisticated process communication mechanism than the binary semaphore needed to implement the critical sections above.

6. Analysis of an Application

We have previously noted that an optimistic approach appears ideal for query dominant systems. In this section we consider another promising application, that of supporting concurrent index operations for very large tree structured indexes. In particular, we examine the use of an optimistic method for supporting concurrent insertions in B-trees (see [2]). Similar types of analysis and similar results can be expected for other types of tree-structured indexes and index operations.

One consideration in analyzing the efficiency of an optimistic method is the expected size of read and write sets, since this relates directly to the time spent in the validation phase. For B-trees, we naturally choose the objects of the read and write sets to be the pages of the B-tree. Now even very large B-trees are only a few levels deep. For example, let a B-tree of order m contain N keys. Then if $m = 199$ and $N \leq 2 \times 10^8$, the depth is at most $1 + \log_{100}((N+1)/2) < 5$. Since insertions do not read or write more than one already existing node on a given level, this means that for B-trees of order 199 containing up to almost two hundred million keys, the size of a read or write set of an insertion will never be more than 4. Since we are able to bound the size of read and write sets by a small constant, we conclude that validation will be fast, the validation time essentially being proportional to the degree of concurrency.

Another important consideration is the time to complete the validation and write phases as compared to the time to complete the read phase (this point was mentioned in Section 4). B-trees are implemented using some paging algorithm, typically least recently used page replaced first. The root page and some of the pages on the first level are normally in

¹This possibility was pointed out by James Saxe.

primary memory; lower level pages usually need to be swapped in. Since insertions always access a leaf page (here, we call a page on the lowest level a leaf page), a typical insertion to a B-tree of depth d will cause $d-1$ or $d-2$ secondary memory accesses. However, the validation and write phases should be able to take place in primary memory. Thus, we expect the read phase to be orders of magnitude longer than the validation and write phases. In fact, since the "densities" of validation and write phases are so low, we believe that the serial validation algorithms of Section 4 should give acceptable performance in most cases.

Our final and most important consideration is determining how likely it is that one insertion will cause another concurrent insertion to be invalidated. Let the B-tree be of order m (m odd), have depth d , and let n be the number of leaf pages. Now, given two insertions I_1 and I_2 , what is the probability that the write set of I_1 intersects the read set of I_2 ? Clearly this depends on the size of the write set of I_1 , and this is determined by the degree of splitting. Splitting occurs only when an insertion is attempted on an already full page, and results in an insertion to the page on the next higher level. Lacking theoretical results on the distribution of the number of keys in B-tree pages, we make the conservative assumption that the number of keys in any page is uniformly distributed between $(m-1)/2$ and $m-1$.² We also assume that an insertion accesses any path from root to leaf equally likely. With these assumptions we find that the write set of I_1 has size i with probability:

$$p_S(i) = \left(\frac{2}{m+1}\right)^{d-1} \left(1 - \frac{2}{m+1}\right).$$

Given the size of the write set of I_1 , an upper bound on the probability that the read set of I_2 intersects the sub-tree written by I_1 is easily derived by assuming the maximal number of pages in the sub-tree, and is:

$$p_I(i) < \frac{m^{i-1}}{n}.$$

Combining these, we find the probability of conflict p_C satisfies:

²This is a conservative assumption since it predicts storage utilization of 75%, but theoretical results do exist for storage utilization [16], which show that storage utilization is about 69%. Since nodes are on the average emptier than our assumption implies, this suggests that the probability of splitting we use is high.

$$p_C = \sum_{1 \leq i \leq d} p_S(i) p_1(i) \\ < \frac{1}{n} \left(1 - \frac{2}{m+1}\right) \sum_{1 \leq i \leq d} \left(\frac{2m}{m+1}\right)^{i-1}$$

For example, if $d = 3$, $m = 199$, and $n = 10^4$, we have $p_C < .0007$. Thus, we see that it is very rare that one insertion would cause another concurrent insertion to restart for large B-trees.

7. Conclusions

A great deal of research has been done on locking approaches to concurrency control, but as noted above, in practice two control mechanisms are used: locking and backup. Here we have begun to investigate solutions to concurrency control that rely almost entirely on the latter mechanism. We may think of the optimistic methods presented here as being orthogonal to locking methods in several ways:

- In a locking approach, transactions are controlled by having them wait at certain points, while in an optimistic approach, transactions are controlled by backing them up.
- In a locking approach, serial equivalence can be proved by partially ordering the transactions by first access time for each object, while in an optimistic approach, transactions are ordered by transaction number assignment.
- The major difficulty in locking approaches is deadlock, which can be solved by using backup; in an optimistic approach, the major difficulty is starvation, which can be solved by using locking.

We have presented two families of concurrency controls with varying degrees of concurrency. These methods are definitely superior to locking methods for systems where transaction conflict is highly unlikely. Examples include query dominant systems and very large tree structured indexes. For these cases, an optimistic method will avoid locking overhead, and may take full advantage of a multiprocessor environment in the validation phase using the parallel validation techniques presented. Some techniques are definitely needed for determining all instances where an optimistic approach is better than a locking approach, and in such cases, which type of optimistic approach should be used.

A more general problem is the following: consider the case of a data base system where transaction conflict is rare, but not rare enough to justify the use of any of the optimistic approaches presented here. Some type of generalized concurrency control is needed that

provides "just the right amount" of locking versus backup. Ideally, this should vary as the likelihood of transaction conflict in the system varies.

References

1. Astrahan et al. System R: Relational Approach to Database Management. *ACM Trans. Database Systems* 1, 2 (June 1976), 97-137.
2. Bayer, R. and McCreight, E. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica* 1, 3 (1972), 173-189.
3. Bayer, R. and Schkolnick, M. Concurrency of Operations on B-trees. *Acta Informatica* 9, 1 (1977), 1-21.
4. Ellis, C. S. Concurrent Search and Insertion in 2-3 Trees. 78-05-01, University of Washington, Department of Computer Science, May, 1978.
5. Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. The Notions of Consistency and Predicate Locks in a Database System. *Comm. ACM* 19, 11 (November 1976), 624-633.
6. Gray, J. Notes on data base operating systems. *Lecture Notes in Computer Science* 60: *Operating Systems*, Ed. R. Bayer, R. M. Graham, and G. Seegmuller, Springer-Verlag, Berlin, Germany, 1978, pp. 393-481.
7. Kung, H.T. and Lehman, P.L. A Concurrent Database Problem: Binary Search Trees. Technical report CMU-CS-79-145, Dept. Computer Science, Carnegie-Mellon University. Also to appear in *ACM Trans. Database Systems*.
8. Kung, H.T. and Papadimitriou, C.H. An Optimality Theory of Concurrency Control for Databases. *Proc. ACM SIGMOD 1979 International Conference on Management of Data*, May, 1979, pp. 116-126.
9. Lamport, L. Towards a Theory of Correctness for Multi-user Data Base Systems. CA-7610-0712, Massachusetts Computer Associates, Inc., October, 1976.
10. Lehman, P. L. and Yao, S. B. Efficient Locking for Concurrent Operations on B-Trees. Submitted for publication.
11. Miller, R.E. and Snyder, L. Multiple Access to B-trees. *Conf. Infol. and Syst.*, March, 1978. Preliminary version.
12. Papadimitriou, C.H., Bernstein, P.A. and Rothnie, J.B. Computational Problems Related to Database Concurrency Control. *Conf. on Theoretical Computer Science*, University of Waterloo, 1977, pp. 275-282.
13. Papadimitriou, C.H. Serializability of Concurrent Updates. Harvard University. To appear in *Journal of the ACM*.
14. Samadi, B. B-trees in a System with Multiple Users. *Information Processing Letters* 5, 4 (October 1976), 107-112.

15. Stearns, R.E., Lewis, P.M. II and Rosenkrantz, D.J. Concurrency Control for Database Systems. *Proc. Seventh Symp. Foundations of Computer Science*, 1976, pp. 19-32.
16. Yao, A. On random 3-2 trees. *Acta Informatica* 2, 9 (1978), 159-170.

ALL INFORMATION CONTAINED
HEREIN IS UNCLASSIFIED
DATE 01-10-2001 BY 60322
AUTHORITY 50101

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) CMU-CS-79-149	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) ON OPTIMISTIC METHODS FOR CONCURRENCY CONTROL.		5. TYPE OF REPORT & PERIOD COVERED (9) Interim Repts.
7. AUTHOR(s) (10) H.T. Kung John T. Robinson		6. PERFORMING ORG. REPORT NUMBER
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA. 15213		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370 NSF-MCS 78-236-76
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA. 22217		10. PROGRAM ELEMENT PROJECT, TASK AREA & WORK UNIT NUMBERS (11) OCT 79
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (12) 202		12. REPORT DATE May 1979
		13. NUMBER OF PAGES 19
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE 403081
16. DISTRIBUTION STATEMENT (of this Report)		
<div style="border: 1px solid black; padding: 5px; text-align: center;"> DISTRIBUTION STATEMENT A Approved for public release; Distribution Unlimited </div>		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
Approved for public release; distribution unlimited		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6001

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403081

14